

Dr. Pine would like us to look at this paper,

**Developing a modern data workflow for living data**

https://www.biorxiv.org/content/early/2018/06/12/344804

*Some points from the publication:*

1) perform quality assurance and control  
-For water quality, the MySQL database has a built in QA/ QC procedure, we are also working with Ben T to create code to check for small moving averages

-For all other oyster data collection, there is a double entry system, and a double entry code that will check for any inconsistencies

- For the portal project they use “data validation” in excel to have a data range for each column

-continuous integration is a `pull request` and also using Travis

2) import, restructure, version, and archive data

* Some databases Zenodo, Dryad, Fig Share

3) rapidly publish new data in ways that ensure appropriate credit to all contributors

4) automate most steps in the data pipeline to reduce the time and effort required by researchers

ADMIN

1. Set up developer
2. Managing `pull requests` to the master branch
3. Permissions set up
   1. New repositories

4) Merge conflicts

DEVELOPER

1. Create new file
2. Update file
3. Delete a file
4. Pull request
   1. Email
   2. Merge conflicts

5) Syncing branch from master

**Meeting notes 8/2/2018**

1. Creating a document for both admin and developer
   1. With checklists, admin and developers and links back to the workflow section
   2. Maybe new names for the roles of admin and developer, **developer**, collaborator
2. Move all of the links and videos for additional training and resources at the end
3. Finish the sections, and will decide workflow order
4. Sarah- refine and test then decide order
5. Mel- adding merge conflicts info, procedures, <- screenshots

7.Travis- will use with functions, not there yet

**What is Github?**

Github is a online portal that allows for developers to store code, documents, images, and videos that enables a version control for any change to the repository. Version control means that for every change or “version”, there is a record of who submitted the change, when , and what exactly they changed.

Version control systems keep these revisions straight, storing the modifications in a central repository. This allows developers to easily collaborate, as they can download a new version of the software, make changes, and upload the newest revision. Every developer can see these new changes, download them, and contribute.

(<https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/>)

There is a good introductory to Github here:

<https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>

Here is another beginners guide to version control:

<https://www.atlassian.com/git/tutorials>

There is no need to download Github, it is a online website ([www.gitub.com](http://www.gitub.com) ), and can be accessed by any computer that connects to the internet.

Sign up for your own free Github account here, (no need to sign up for the upgraded GitHub account type):

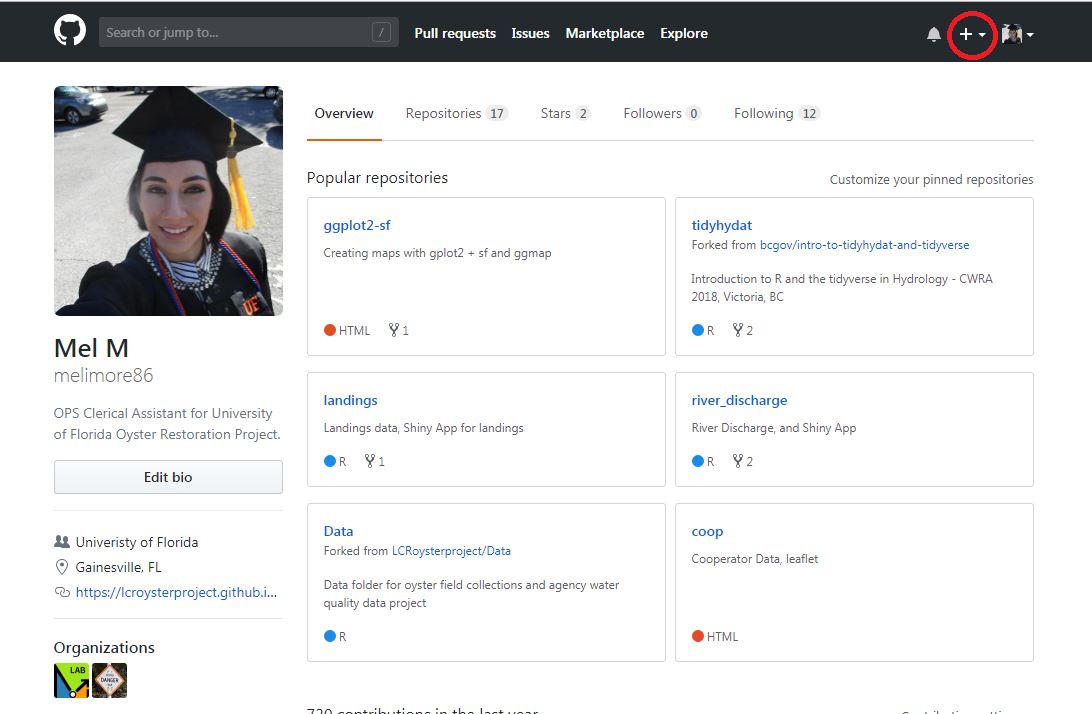
<https://github.com/join>

**What is a repository?**

A repository (usually abbreviated to “repo”) is a location where all the files for a particular project are stored. Each project has its own repo, and you can access it with a unique URL. (<https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/> )

Best practices for repositories, are to create a new repository for each project, instead of having one repository with folders for each project. This can avoid confusion in the future, when you are in need to locate certain files or to direct others to clone your repository.

To create a new repository click on the “+” in your account. For this project however, there is a repository already in place.



Repository Figure 1- Basic profile of a developer and admin, where they can create a repository at the top left “+”. The admin will always create the new repository for the project, but a developer can create their own repository for their own personal use.

The link for the project Data page, <https://github.com/LCRoysterproject/Data> .

**What does it mean when we refer to local files/machine?**

* Local is the terminology used to describe files, or folders on your local computer.
* Machine is another term for your personal/work laptop, computer.
* Files from Github can be directly downloaded into your computer either through the Github Portal, or through the Git Bash command line.

**What is Git Bash?**

Git Bash is a command prompt to write out code to submit changes from your local machine to the Github online portal. All commands that can be executed in the command prompt, can also be completed in the Github portal.

Some advantages of using Git Bash:

* Allows submission of changes through RStudio directly into your GtiHub repository.
* Allows for status checks on development branches to master branch.

\* Although, any file that is changed, can be directly uploaded into GitHub instead.

Here is a video on how to integrate Github and R Studio:

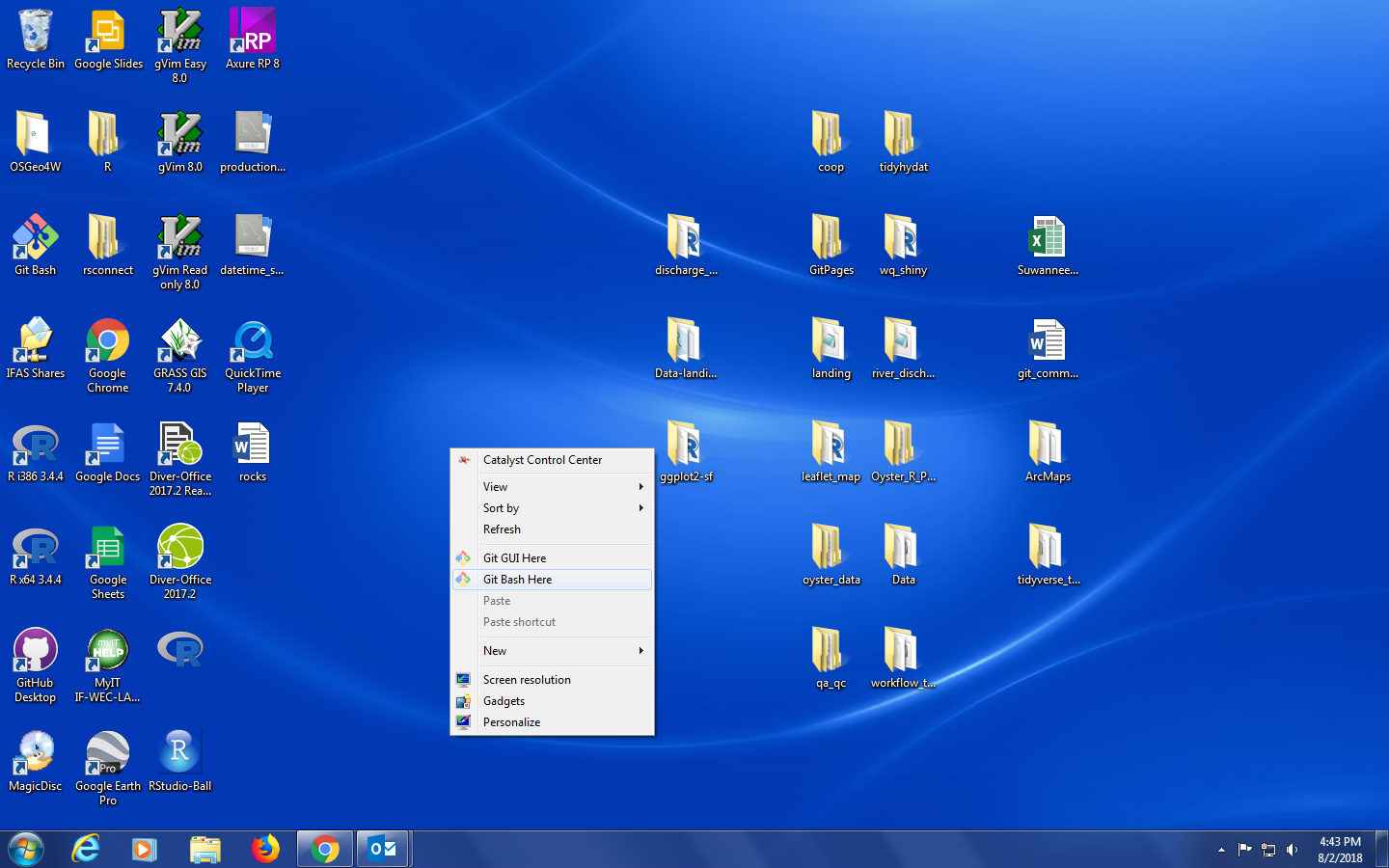
<https://www.youtube.com/watch?v=E2d91v1Twcc>

It is normal practice to use both, Git Bash and Github portal, to clone repositories to your local machine, track your status updates, change files, commit pushes, and do pull requests. It is up to each individual developer on what they prefer, but we will outline both practices in this workflow documentation.

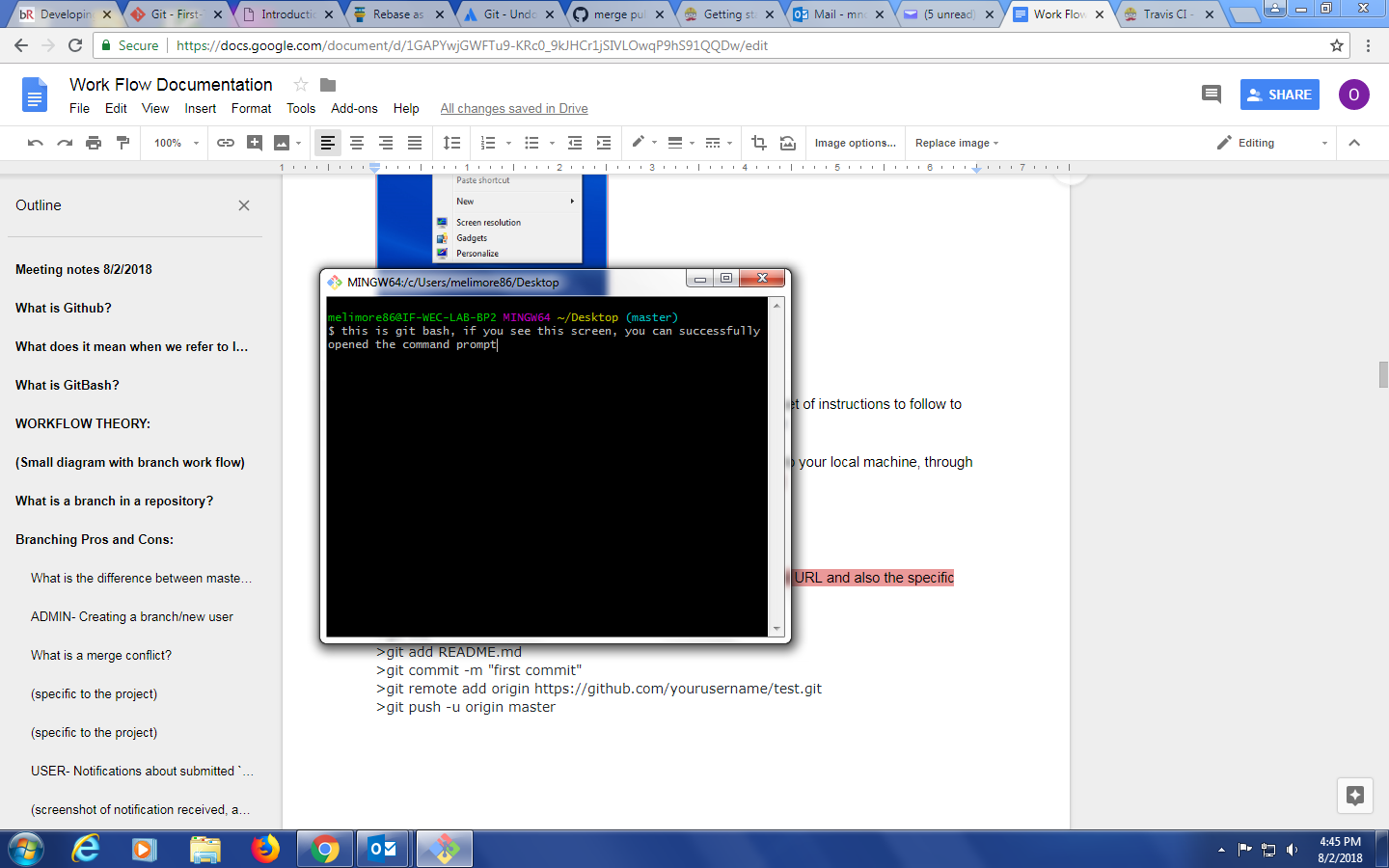
You can download Git Bash here:

<https://git-scm.com/downloads>

Once you download Git Bash, you can right-click on your mouse, to open the Git Bash terminal. If you do not right-click on a already existing cloned folder, you will need to navigate to the folder, in terminal, through a series of commands.



Gtt Bash Figure 1- Right click on the mouse to access the Git Bash Here option.



Git Bash Figure 2- . This is the command prompt of the Git Bash terminal.

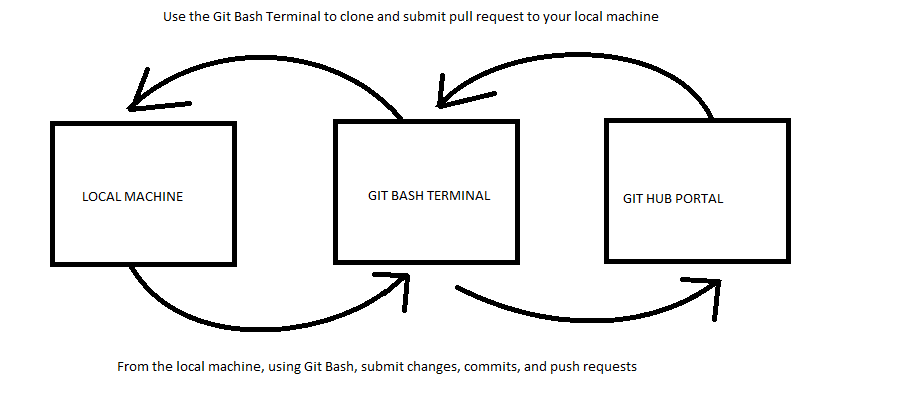
Here is a good reference to GitBash commands:

<https://sklise.com/2012/09/22/introduction-to-git/>

When a repository is created in the GitHub portal, there will be a set of instructions to follow to clone the repository into your local machine. You will use Git Bash to connect to the project Github directly.

The command for cloning the repository is:

git clone https://github.com/LCRoysterproject/Data.git

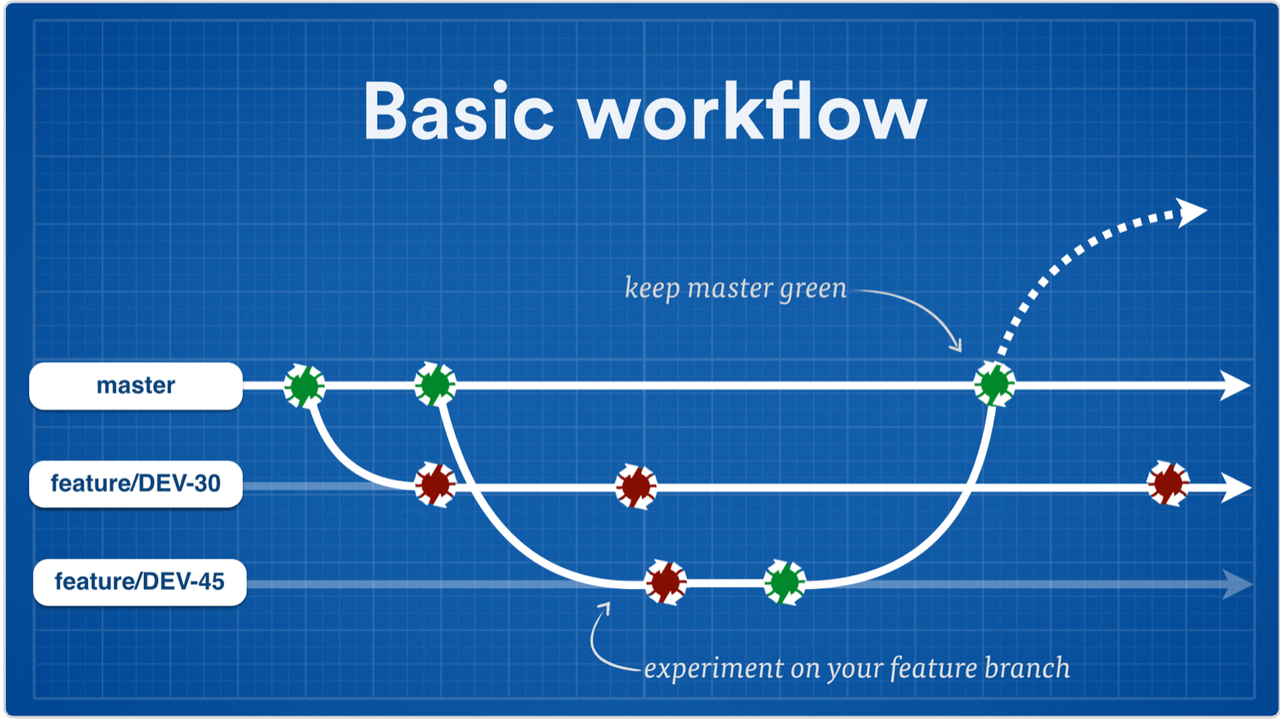


Git Bash Theory Figure 3- Basic workflow of using Git Bash.

* Create or use an existing repository, in this case we will be using https://github.com/LCRoysterproject/Data.git
* Clone the repository to your local machine, new repositories will have Git Bash command steps to initiate the repository to the local machine
* Use the Git Bash terminal to add/ commit/ push changes to the Git Hub portal
* You can choose to delete the local machine copy of the repository, to keep the Git Hub portal repository as the most up to date version, instead of having out of date local copies, this can avoid confusion of which repository has the latest changes
* If a local repository is kept on the local machine, use a pull request to update the local repository to the latest version of the Github Portal

**WORKFLOW THEORY:**

Using a series of developer development branch to create, edit, and modify files to the master branch, for production.



(picture credit https://www.atlassian.com/continuous-delivery/continuous-delivery-workflows-with-feature-branching-and-gitflow)

Workflow Theory Figure 1- Basic branch workflow indicating that developer branches will have changes/commits (red circles) and will eventually be up to date with the master branch (green circles) or merge the changes into the master branch.

**What are the roles of the collaborators in the project repository?**

**Admin-**

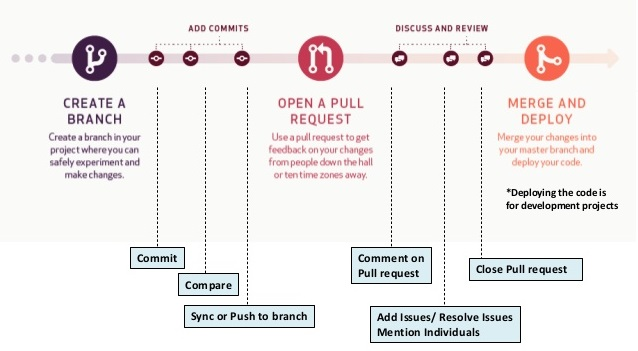
* Monitors the `master` branch, usually 1-2 people
* Creates development branch for each developer, and deletes branches when developer is no longer working on project
* Reviews all `pull requests` from each developer, chooses whether to approve, change, or deny the `pull request`
* Creates permissions for the repository

**Developer-**

* Utilizes developer specific branch to modify, delete, or change code/ text
* Works independently of other branches including the `master` branch
* Submits `pull requests` to the `master` branch to merge changes
* Checks emails frequently once a `pull request` is submitted

**What is a branch in a repository?**

Branches are separate “clones” of the `master` branch, that can `pull` changes from the `master` or commit `pull requests`. Branches are located within one repository.   
  
Through a series of developer specific branches, each developer will submit their changes/ modifications to the master branch, that will be approved by the admin (Mel and/or Bill Pine). Pull requests can be approved, commented on, or denied. All pull requests will have thorough documentation for each submission. Branches will not be deleted. Each branch must submit a pull request from the master -> branch, prior to any pull requests. Each step will be explained in further detail.

****

(picture credit <http://www.smlcodes.com/tutorials/github-tutorial/>)

Branch Figure 1- A basic workflow working with branches. Admin will create the branch for the developer, where the developer will commit their changes, and submit a pull request from the developer branch -> master branch. Pull requests will have comments from the admin, where the issues can be resolved (if any), and the pull request will be approved, and the pull request will be closed.

**Branching Pros and Cons:**

## Pros:

* Keeps all of the work being done around a project in one place
  + No confusion with forks, or forking
* All collaborators can push to the same branch to collaborate on it
  + All collaborators will be able to view the master branch pull request after they are approved
* There's only one Git remote to deal with

### Cons:

* Branches that get abandoned can pile up more easily
  + Not in our case, since each developer will have their own branch, and developers that are not on the project any more, will have their branches deleted
* Your team contribution process doesn't match the external contributor process
  + Both processes are indeed different
* You need to add team members as contributors before they can branch
  + This will be the responsibility of the Admin

**What is the difference between master branch, and developer development branches?**

The master branch can be considered the production/live branch. The production branch is where all reviewed code and documents submitted to for review, and all approved changes will be live. The master branch is protected, and should only be handled/reviewed by the Admin Developers will not work directly in this branch.



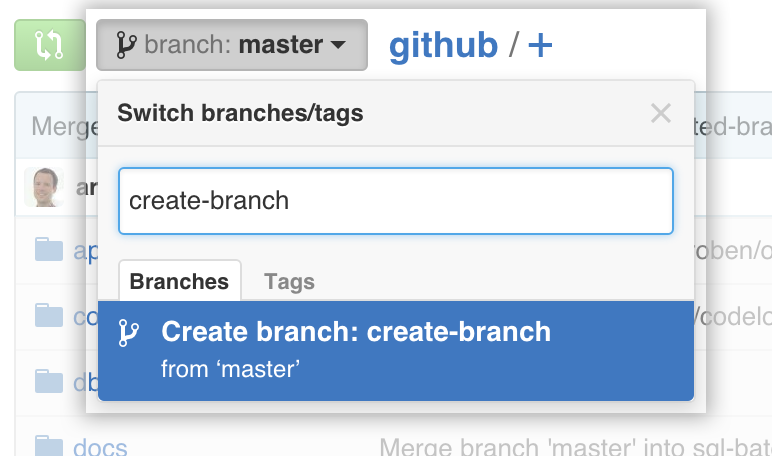
The developer development branch is for testing and development of code, or edits of documents. Each developer will have their own development branch, in which they will work on solely on. Developers can work on their own branch, with no Admin monitoring. Developers are free to make and changes/modifications/edits to their own branch. A new branch will be created for each collaborator.

**ADMIN- Creating a new developer branch**

Setting up the developer involves creating a new branch for each developer. The Admin is the only allowed personnel to create/delete branches.

Each developer will need a new branch, in which they will work on solely.

When a new branch is created in a repository, an environment is created to test new ideas. Changes that are made on a development branch doesn’t affect the `master` branch, so the developer is free to make and commit changes to their own branch. The `master` branch is safe in the knowledge that a development branch won't be merged until it's ready to be reviewed by the ADMIN. (<https://guides.github.com/introduction/flow/>)



Creating branch Figure 1- The branches will be created from the `master` branch. At the time the developer branch is created, the new branch will be an exact clone of the `master` branch.

<USERNAME> - DEVELOPMENT

**ADMIN- Managing `pull request` and `merges`**

The master branch will receive the `pull request` from the developer . An Admin will be selected to review the changes prior to merging to the master branch. The repository will check to automatically correct any `merge conflicts`

**What is a merge conflict?**

A `merge conflict` occurs when the branch that is submitting a `pull request` to the master branch was not originally up to date with the `master branch. This will cause Github to say: 

Github will not know which code/ text to change if both files started off fundamentally different. Here is an example.

Imagine a text file in the master repository, starting like this:

-Master branch:

----------

This is a brown cat.

----------

Now imagine, that a development branch has an older version of the text file, like this:

-Development branch:

----------

This is a cat.

----------

Now lets say that the development branch starts working off this file, without the “brown”, and makes changes:

-Development branch:

----------

-This is a cat. This cat is also fluffy.

----------

Now this development branch wants to send a `pull request` from the development branch -> master branch. There will be a `merge conflict`. The development branch must have the most up to date version of the file they are trying to change, before making any changes.

Now lets theoretically, make these changes again, but with the development branch using the most up to do date version of the text file.

-Master branch:

----------

This is a brown cat.

----------

Now imagine, that a development branch has the most up to date version of the file, like this:

-Development branch:

----------

This is a brown cat.

----------

Now lets say that the development branch starts working off the most current file version:

-Development branch:

----------

This is a cat. This cat is also fluffy.

----------

This `pull request` can now be merged because the development branch was using the most up to date version of the file prior to the `pull request`.

Any `merge conflicts` will have to be review by the Admin, and resolved by the Admin.

(<https://www.git-tower.com/learn/git/ebook/en/command-line/advanced-topics/merge-conflicts> )

**What is a pull request?**

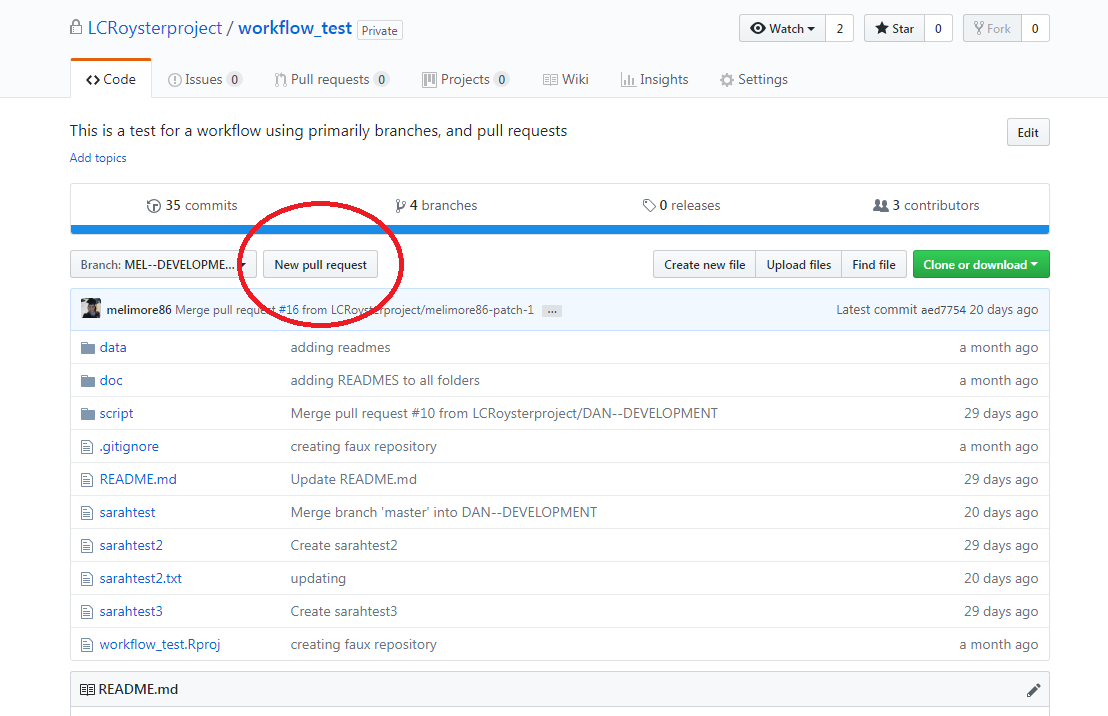
Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a `pull request` is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

Note: When working with pull requests, keep the following in mind:

* If you're working in the [shared repository model](https://help.github.com/articles/about-collaborative-development-models), we recommend that you use a topic branch for your pull request. While you can send pull requests from any branch or commit, with a topic branch you can push follow-up commits if you need to update your proposed changes.
* When pushing commits to a pull request, don't force push. Force pushing can corrupt your pull request.

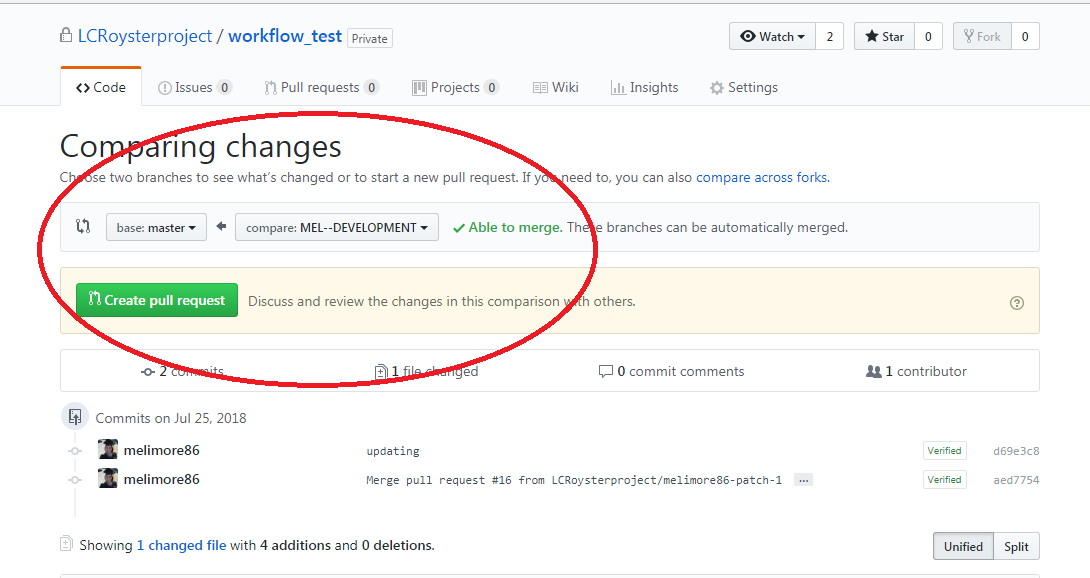
After initializing a pull request, you'll see a review page that shows a high-level overview of the changes between your branch (the compare branch) and the repository's base branch. You can add a summary of the proposed changes, review the changes made by commits, add labels, milestones, and assignees, and @mention individual contributors or teams. (<https://help.github.com/articles/about-pull-requests/>)

 **Each developer will have their own branch, in which they will be the only developer to work in it. Any change, update/change/modification**, will be sent to the master branch through a pull request from the developer branch -> master branch.   
Click on “New pull request”.



Pull Request Figure 1- Screenshot of where to start a New Pull Request.



  
 Pull Request Figure 2- Screenshot showing how to compare the developer branch to the master branch. The developer branch is `Able to merge` because the developer branch was originally up to date with the master branch.

If there are any merge conflicts, contact your admin.

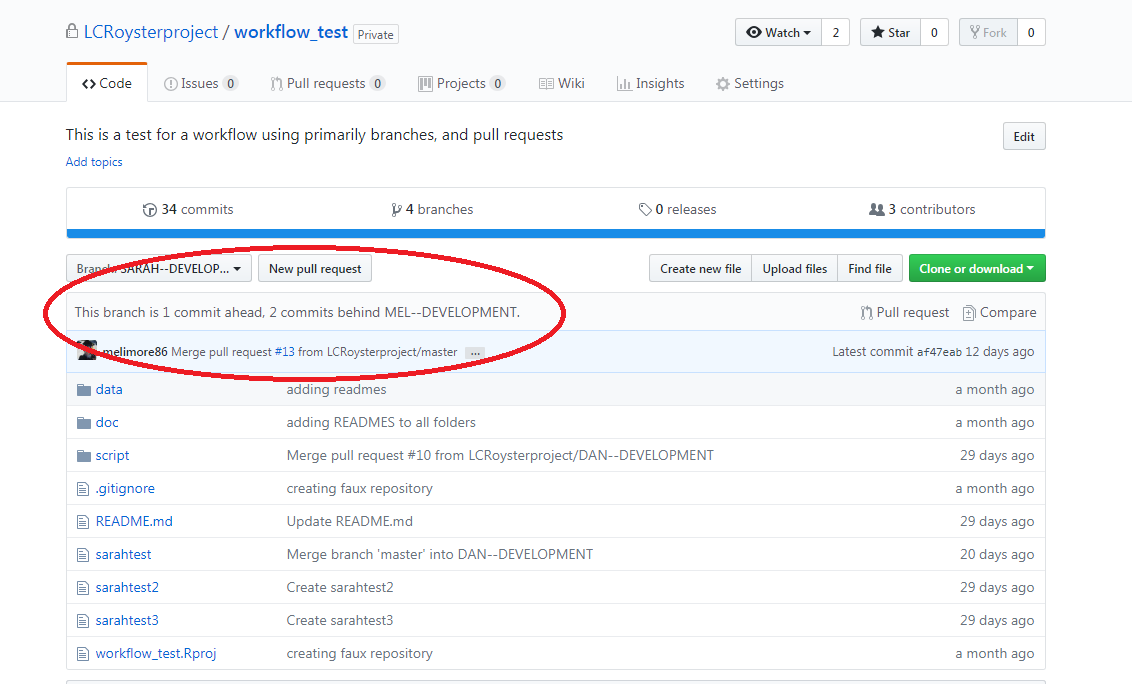
It is imperative that the `pull request`, to submit changes to the production master branch, be from the **development branch-> master**. This means that the developer is trying to merge their changes to the master branch.

If your developer branch has no merge conflicts, there will be a message “Able to merge”. This means that the changes you are submitting were created using the latest version of the master branch, prior to creating the changes.

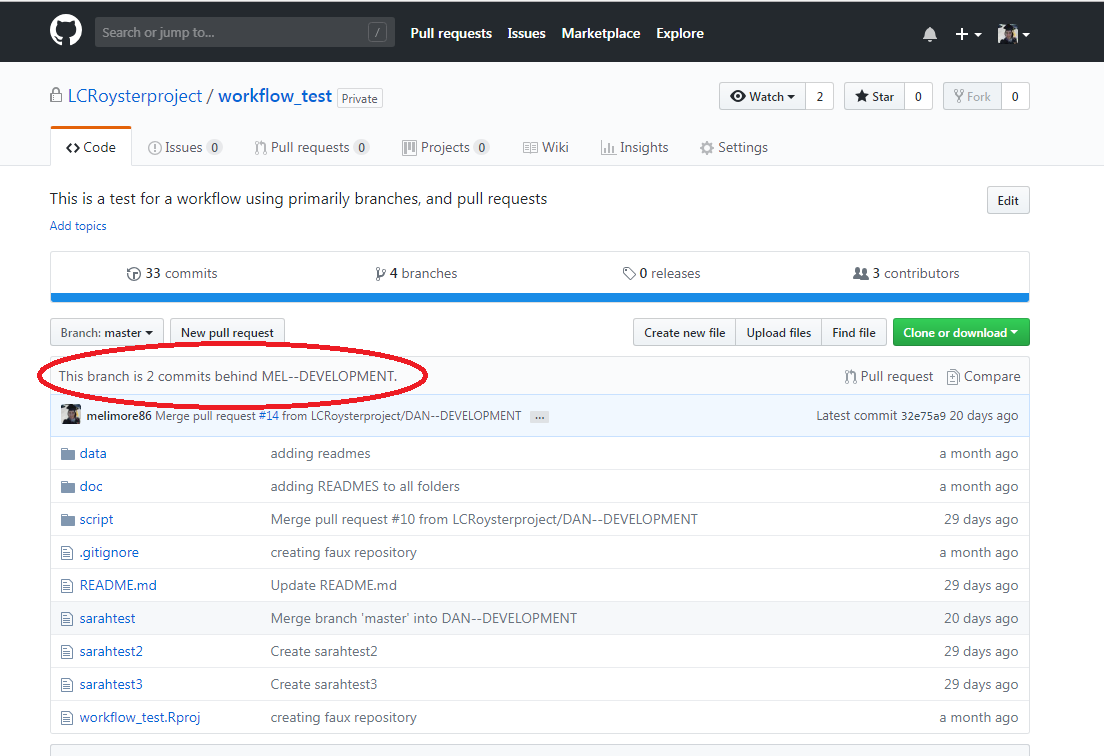


Here are a few ways to check if the developer branch is up to date with the master branch.

In the front of the developer branch, there will be a message on how many commits any branch is ahead, including developers and master branch.



Pull Request Figure 3- A developer can view if they are ahead or behind any of the branches in the repository.



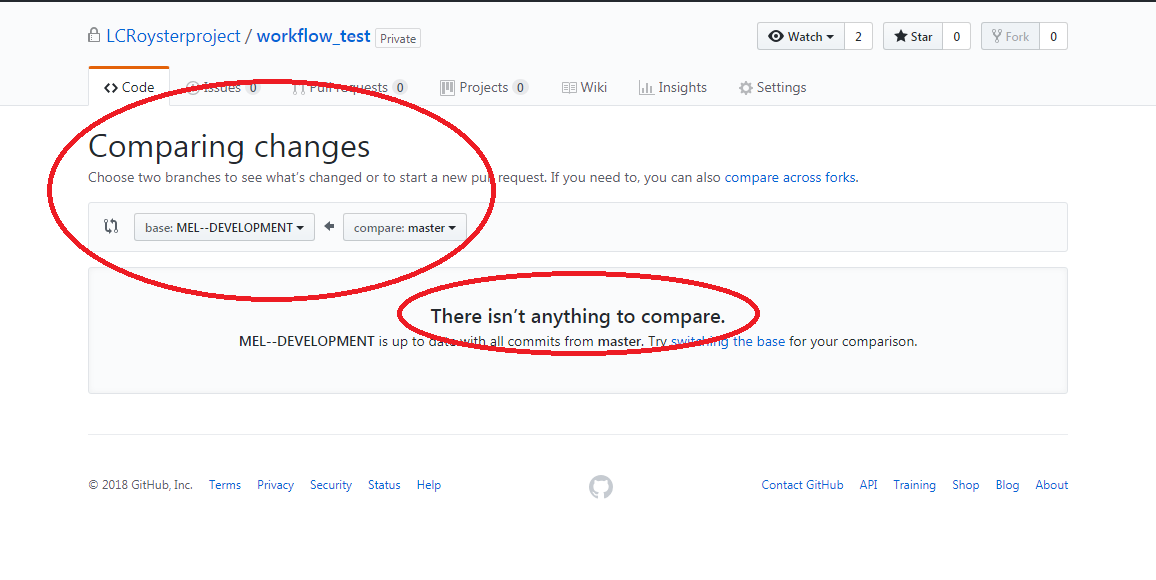
Pull Request Figure 4- The developer will not be able to make any changes to the master branch, but a developer can click on the master branch, to view if the master branch is ahead or behind of any of the branches in the repository.



A `pull request` can also be set from the **master-> development branch**, so that the development branch will have no merge conflicts when they submit a pull request ( development branch -> master).

This does not need Admin approval, and can be done at any time, **BUT MUST** be done **EVERY TIME** that the developer wants to continue working on their branch, that is not up to date with the master. Use the previous steps to check if the developer branch is the same as the master branch.

If you see a message “There isn’t anything to compare”, then your developer branch is up to date with the master branch, and you can start making changes to your developer branch.



Pull Request Figure 6- Screenshot of the commit status of the master branch to the developer branch.



**ADMIN- Permissions Set Up**

Permissions for the repository are set up by the Admin. Permissions can be set up in Settings -> Branches -> Protected branches. From there the Admin will choose to protect the master branch, by clicking the dropdown box “Choose a branch” and selecting master. There will be many settings for the repository, but the minimum that should be selected are :

* Protect this branch
  + Require pull request reviews before merging
    - Required approving reviews: (at least 1)
* 2) Included administrators

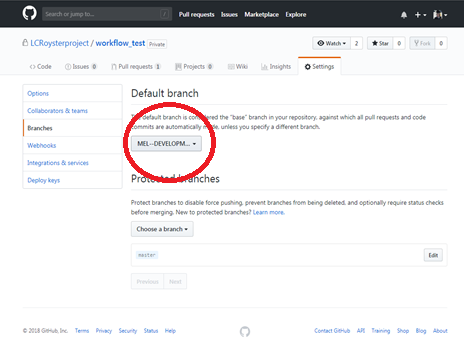
Of course more protection options can be placed, but these options are enough to protect the branch. With each protection option, there will be a description of what the protection option does.

Below are all of the permissions that are in the “Branch Protection”, and the highlighted ones are the minimum permissions for the branch.

* Protect this branch. (Disables force-pushes to this branch and prevents it from being deleted.)
* Require pull request reviews before merging (When enabled, all commits must be made to a non-protected branch and submitted via a pull request with the required number of approving reviews and no changes requested before it can be merged into master.)
* Required approving reviews: 1 (at least one)
* Dismiss stale pull request approvals when new commits are pushed. (New reviewable commits pushed to a branch will dismiss pull request review approvals.)
* Require review from Code Owners (Require an approved review in pull requests that include files with a [designated code owner](https://help.github.com/articles/about-codeowners).)
* Restrict who can dismiss pull request reviews. (Specify people or teams allowed to dismiss pull request reviews.)
* Require status checks to pass before merging (Choose which [status checks](https://developer.github.com/v3/repos/statuses/) must pass before branches can be merged into master. When enabled, commits must first be pushed to another branch, then merged or pushed directly to master after status checks have passed.)
* Require branches to be up to date before merging (This ensures the branch has been tested with the latest code on master. This setting will not take effect unless at least one status check is enabled.)
* Require signed commits.(Commits pushed to this branch must have verified signatures.)
* Include administrators. (Enforce all configured restrictions for administrators.)
* Restrict who can push to this branch.( Specify people or teams allowed to push to this branch. Required status checks will still prevent these people from merging if the checks fail.)

**Developer- Default branch**  
Part of the workflow is to change each developer branch, to their default branch in the repository. Select the default branch with your name. If your branch is not there, let the Admin know.

**Go to Settings-> Branches-> Default Branch**

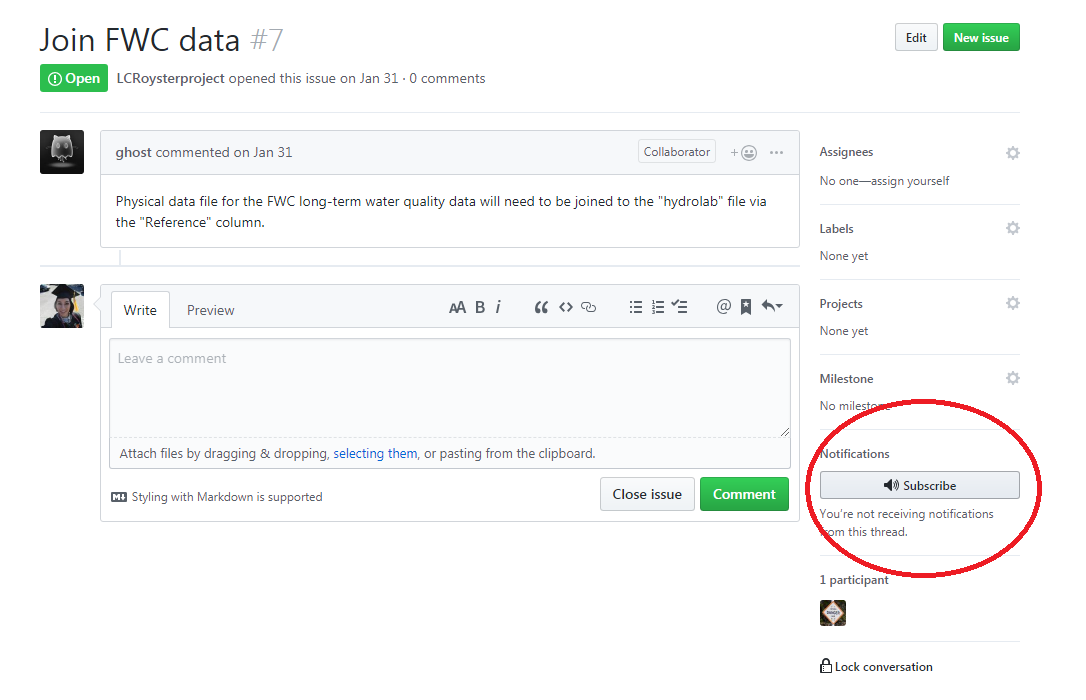


Default Branch Figure 1- When logged into your username, you need to select your developer branch, as the default branch. This will avoid confusion in the future. Change the drop down where the red circle is, if your username is not here, let the admin know.

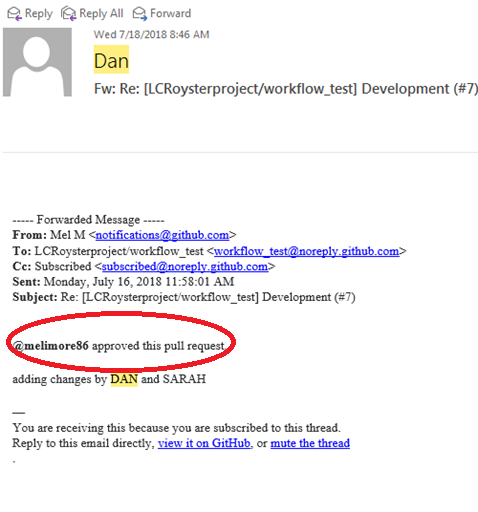
**Developer- Notifications about submitted `pull requests`**

Developers will need to be have their email available to check on `pull request` activities. An email will be sent to the developer that indicates the status of their pull request to the `master` branch. Whenever the Admin submits a change to the developers’`pull request`, the developer will receive an email with the `pull request` outcome. Only the branch that submits the `pull request` will receive any notifications about its outcome, so other developer will not receive an email.

Another option is also subscribe to notifications about the pending `pull request`.



Notification Figure 1- Clicking on the Subscribe button will send a notification about the status of the pull request.



Notification Figure 2- Email sent from the ‘pull request’ update

**Reasons why pull requests are not approved**

* 1. Naming conventions are not appropriate( see Naming convention guidelines below)
  2. Wrong branch submitting pull request i.e. developer submitting pull request from another developer’s branch
  3. Ineffective code or text i.e code or text tested and not approved

**Naming Conventions for files**

-No spaces in file names ALWAYS use \_ (underscore)

- date first yyyymmdd, and then category/gear type ( i.e transect, quadrat, biomass, tong), following site if applicable

-i.e 20180802\_transect.csv, text file

-i.e 20180809\_salinity\_allsites.png, figure file, can use “allsites” for sites 1-9, but otherwise label the sites, 20180809\_salinity\_123.png

- No camels (upper and lower, ALWAYS lowercase)

- Invalid characters, i.e $,#,!, &,(),@,%,^,\*,”

**Naming Conventions for folders**

fig-figures

doc- documents, text files

data- collected data, usually in .csv or .xlsx

script- for R code, ‘data’ folder must be located inside for a R project  
compare- folder where script and files will be stored for future comparison

reconcile- folder for cleaning scripts and files can be stored, separate from compared folder

* Admin will always reject the files or folders unless they are in proper format.

**Developer- Git Bash commands for specific branches**

To make sure that developer is editing their specific branch, a series of commands needs to be entered to make the changes are submitted to the correct branch.

Clone repository to your clone machine

*git clone url*

Check branches of the repository (mandatory)

*git branch –a*

Checkout on USER DEVELOPMENT branch (mandatory)

*git checkout -b USER DEVELOPMENT*

Add changes to be pushed(mandatory)

*git add nameoffile*

or

Add changes to the whole branch

*git add -A*

Add a message for the commit (mandatory)

*git commit –m “message you are adding”*

Now push your changes

*git push origin USER DEVELOPMENT*

Now the branch USER DEVELOPMENT is updated. Tests can be done on this new code in the DEVELOPMENT branch, and once it is verified, a pull request can be used to compare across branches before they are merged.

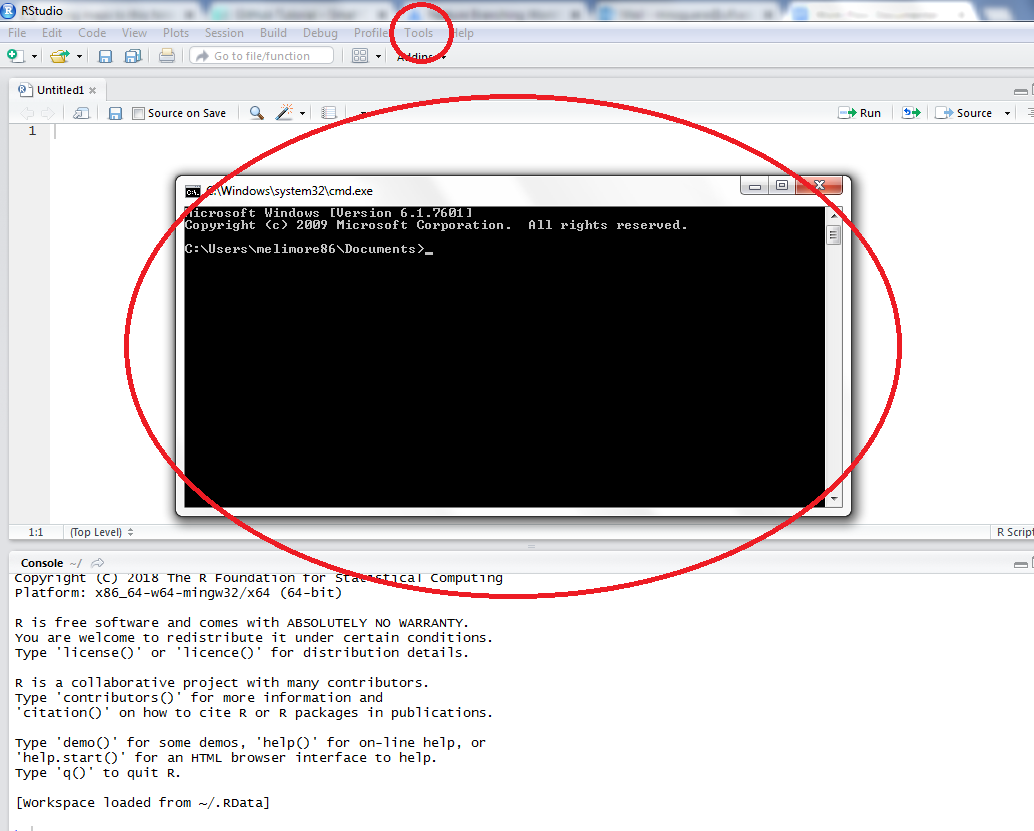
**Developer- R Studio and branches**

RStudio can’t create branches directly, so you need to either:

* create them in GitHub and pull the changes in your repository;
* create them from the Shell (Tools > Shell) and type:
  + git checkout -b USER DEVELOPMENT

<http://r-bio.github.io/intro-git-rstudio/>

**Developer- Updating developer branch on local machine**



R Studio branches Figure 1- Shell (Tools > Shell) and type: git checkout -b USER DEVELOPMENT, a prompt will appear where you can select the branch and then follow the steps in “**Developer- Git Bash commands for specific branches”**

**Admin- Creating an R Project**

Each repository will have an R project associated with it.

1. Set up an R project in R studio

a. File -> new project -> existing directory

b. Find the location of your Data folder: for me it’s Desktop/git/Data

2. If you already have an Rproject and you want to connect it to a github repository:

a. In Rstudio go to Tools -> Version Control -> Project Setup

b. I have not actually done this before but this website may be helpful:<https://support.rstudio.com/hc/en-us/articles/200532077?version=1.0.143&mode=desktop>

**Data Entry - Spreadsheets setup**

1) You need a value for cells with missing data. The value needs to be consistent with the data type for that column. **For numeric cells USGS uses -999 or another value way out of the range of possible values**. For missing time values I used 0:01. **For missing text N\_A is preferred**.

2) NULL values are acceptable, but impossible to put into a blank cell in Excel. Geodatabases and other database tables will accept Null values.

3) I have modified column names to be ready to use in GIS. **Column names should be no longer than 10 characters**.

4) I have added two columns utm\_x and utm\_y. These columns used formulas to correct the data in utm\_east, and utm\_north, those columns need to be integer values.

5) utm\_x cells contain: =IF(LEFT(N2,4)="17R0",RIGHT(N2,6)\*1,N2\*1) and utm\_y cells contain: =O2\*1 where N is the utm\_east column and O is the utm\_north column.

6) For data entry utm\_eastings do not need the leading 17R0

7) The biggest name changes are: Reef\_elem\_rough column became est\_rf\_elm (estimated reef element), and Short\_cm became height\_cm.

8) Text values should be left justified, numeric, date, and time values should be right justified.

9) **Each column should be formatted for the type of values it contains**

10) I would not use leading zeros for months or days as was done on the raw\_data\_1 sheet.

11) Year, month, and day columns I would format as a number with 0 decimals

12) Make sure that the pick list columns are also correctly formatted for value type.

13) The final cleaned table for GIS use should not contain any formulas.

14) Columns to be **lowercase**, but text values are **uppercase**.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| quad | tran | spat | bird | invert | rock | def | type | just | N/A |
| year | year | year | year | year | year | YYYY | num | right | -9999 |
| month | month | month | month | month | month | MM | num | right | -99 |
| day | day | day | day | day | day | DD | num | right | -99 |
| strt\_time | strt\_time | strt\_time | strt\_time | strt\_time | strt\_time | Military  00:00 | num | right | 0:01 |
| end\_time | end\_time | end\_time | end\_time | end\_time | end\_time | Military  00:00 | num | right | 0:01 |
| locality | locality | locality | locality | locality | locality | AB | char | left | N\_A |
| bar | bar | bar | - | - | - | XX | num | right | -999 |
| - | - | - | - | - | long\_cm | XX.X | num | right | -999 |
| - | - | - | - | - | height\_cm | XX.X | num | right | -999 |
| - | - | - | - | - | wide\_cm | XX.X | num | right | -999 |
| reef\_loc | reef\_loc | - | - | - | reef\_loc | A | char | left | N\_A |
| utm\_east | utm\_east | utm\_east | - | utm\_east | utm\_east | (X)^6 | num | right | -999 |
| utm\_north | utm\_north | utm\_north | - | utm\_north | utm\_north | (X)^7 | num | right | -999 |
| recorder | recorder | recorder | recorder | recorder | recorder | AB | char | left | N\_A |
| utm\_x | utm\_x | utm\_x | - | utm\_x | utm\_x | (X)^6 | num | right | -999 |
| utm\_y | utm\_y | utm\_y | - | utm\_y | utm\_y | (X)^7 | num | right | -999 |
| quad | tran | spat | bird | invert | rock | def | type | just | N/A |
| time\_zn | time\_zn | time\_zn | time\_zn | time\_zn | time\_zn | EST UTC | char | left | N\_A |
| station | station | - | - | - | - | ABCX | char | left | N\_A |
| quad\_id | - | - | - | - | - | date\_station\_ | char | left | N\_A |
| live\_dead | - | - | - | - |  | L  D | char | left | N\_A |
| size\_cm | - | - | - | - | - | XX.X | num | right | -999 |
| count | - | - | - | - | - | XX | num | right | -999 |
| treatment | treatment | - | - | - | - | control  rock | char | left | N\_A |
| - | tran\_length | - | - | - | - | X + 2.5=  “Length”  XX.XX | num | right | -999 |
| - | replicate | - | - | - | - | X | char | left | N\_A |
| - | - | - | - | - | est\_rf\_elm | X | num | right | -999 |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

**(Example of the different/ 1-2 sampling naming convention standards for the columns)**

Versioning in Git for each

The NAs

NULL for MySQL

NAs will not work with MySQL

Cannot be empty for ArcMap

-999 for numeric

N\_A, not NAs

TOAD DMS, to a data column approving